uc3m | Universidad Carlos III de Madrid

Bachelor's Degree in Computer Engineering
2017/2018

Bachelor Thesis

Programming language identification using machine learning

Francisco Javier Honduvilla Coto

Supervisor: Nerea Luis Mingueza

Leganés, Marzo del 2018

Abstract

Many developer tools such as code search and source code highlighting require to know the programming language each given file is written in. Another task that also requires knowing the programming language of some source code file is the generation of statistics for code repositories.

The main aim of this project is to build a programming language classifier that could be used in the previously mentioned tasks. This is done mainly by employing machine learning in combination with text classification techniques.

We have developed a source code classifier that has been previously trained and then tested using source code from the Rosetta project dataset [44]. We also measure some metrics such as the time it takes to train the classifier, its accuracy, and time to perform the classification of individual source files.

We also assess the economic impact in this social context, evaluating why a tool like this makes sense for developers.

Resumen

Muchas herramientas diseñadas para programadores, como búsqueda de código o coloreado de sintaxis requieren conocer previamente en qué lenguaje está escrito cada fichero para funcionar correctamente. Generar estadísticas sobre repositorios de código también precisa de esta información.

El principal objectivo de este Trabajo de Fin de Grado es construir un clasificador de lenguajes de programación que pueda ser utilizado en las tareas mencionadas anteriormente empleando aprendizaje automático junto con técnicas de procesamiento de textos.

Hemos desarrollado un clasificador de código fuente que ha sido entrenado y posteriormente evaluado con código fuente del proyecto Rosetta [44]. Tambien hemos medido algunas métricas como el tiempo de entrenamiento de los clasificadores, su precisión, y el tiempo que tardan en clasificar ficheros de código fuente individuales.

Tambien se expone el impacto económico en este contexto social, evaluando por qué esta herramienta es importante para desarrolladores.

Acknowledgements

Thanks to my parents for their lifelong support.

Thanks to all my University friends for all the learning and life experiences we had together. It was crucial to have someone joking around while working after midnight in some University projects!

Thanks to all friends during these last years that supported and helped me in any way, I really appreciate it!

Last, but not least, thanks to my supervisor Nerea for all her ideas, time, and general help, as well as all the trust she gave me in this project.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

The "developer experience", that is, the user experience of the tools used by developers is important for their productivity, ease of use, and general happiness [27].

Being able to automatically classify source code files by programming language aligns extremely well towards a better developer experience, as it can offer:

- **Syntax code highlighting**: Colouring of source code enables developers to grasp the code faster [7]. Usually, that is done by splitting the source code into tokens and mapping each token to a different colour.
- **Improved code search**: The ability to search in a codebase is important to navigate fast through it. Git itself provides some functionality with its `grep` sub-command. Some Git hosting services, such as GitHub [19] have a ElasticSearch-backed search engine [2]. Google Code [14] used to have a really interesting and advanced system that allowed to search using regular expressions while still being performant by using trigram-based indexing [43] and an efficient regular expression library called re2 [42]. Drilling down by specific characteristics, such as by programming language, is an extremely useful feature.

Another use case is performing repository mining to extract statistics from publicly available source code repositories. Some of the possible purposes are:

- Programming language popularity analysis.
- Keyword / sentiment analysis, such as extracting how many profanities programmers make broken down by programming language [5].
- Recruiting: to offer a more tailored offer to developers based on the programming languages they seem to use or have some expertise in.

These are quite relevant issues nowadays, as the open source movement is bigger than ever [12]. Individuals and companies have started realising the multiple benefits of adopting open source software as well as making some of their internal projects open source.

1.2 Initial ideas

Source code identification, like many other classification tasks, seems rather trivial for humans in the majority of cases, however, it's a bit more challenging for machines.

One of the first ideas is using the file extension as a heuristic. It has some challenges described below, so we build the core classifiers without using file extension information. Some of the reasons why we are trying to avoid file extensions are:

- Code can be inlined in another file, such as Markdown for example, lacking the original file extension.
- Several programming languages share the same extension. For example, `.fs` is shared by Forth, F, GLSL, and Filterscript.
- Even if we knew the extension, it could be wrong.

The ideas we had that could help in identifying source code:

- **Character and tokens statistics**: Not all tokens are used equally in every programming language. A contrived example can be the amount of open parenthesis, "(", that we can find in LISP vs the amount we can find in the MIPS assembly.
- **Grammars**: If we wanted to be more accurate, grammar-wise, we could run all the grammars of the languages we are interested in and check which ones end up in a "final state", that is, without errors. Those which end up there are accepted by that grammar and are highly likely valid languages for that grammar.
- **Syntax checkers**: Some programming languages interpreters / compilers have a mode that just runs the Lexer and the Parser and let you know if the tokens and syntax are correct. As we stated before, that doesn't necessarily imply that that code will run without errors (e.g: when using a variable that wasn't declared before in a dynamic language). This technique also has a very big performance disadvantage; if we have to shell out to run this per each language, it's going to take a while to complete.
- **Tool traces**: Some developers leave special comments so-called "magic comments" in their source files as text editor configurations which sometimes indicate unambiguously the programming language that was used.

Some code constructs that can make our task more difficult are:

- Comments. When porting code, it is common to keep the original code in comments to be sure that the translation is as accurate as possible.
- Languages that embed other languages. For example, PHP than can have embedded HTML, which can have JavaScript inside which can have CSS in some strings to apply styles programmatically.
- Quines are programs that when they are executed, produce valid source code for another programming language. There are some very convoluted ones that manage to do a "quine relay" that basically closes the loop.

All of them introduce problems when using statistical methods, as we are identifying the source code as a whole. Another challenge introduced by code comments is that we can only remove them safely when we know which programming language is being used. At the same time, we only know what kind of syntax the comment declaration uses when we know the language.

1.3 Objectives

The main objective of this project is to build a programming language classifier that can perform well over varied fragments of code. We also implement code for the training of a classifier using machine learning techniques. The classifier will be trained and validated using code from the Rosetta project [44], and will have functional and non-functional characteristics that will be described in the next sections.

2 State of the art

Identifying which programming language has been used for a particular source code can be useful for many tasks, like syntax highlighting and formatting. Most software developers go through this kind of tasks on a daily basis.

With the explosion of open source, driven in part by websites like Sourceforge [47] and GitHub [19], as well as the adoption and support by big and small corporations, other use-cases like source code mining, have become more relevant.

Some of the most interesting research papers that were found that explore this topic are:

- "Detecting Programming Language from Source Code Using Bayesian Learning Techniques" extensively explores using Bayesian learning techniques for detection of programming languages [31].
- "What's the code? Automatic Classification of Source Code Archives" uses support vector machines (SVMs) [50].
- "Identifying source code programming languages through natural language processing" experiments with using natural language processing techniques for this problem [11].

"Source Code Author Identification Based on N-Gram Author Profiles" is not completely related to programming language identification but it is also interesting as it investigates on identifying the author of a given source code using n-grams.

In this chapter we review the background of some important pieces of software and Computer Science concepts that are related to this problem, such as Git [17], compiler theory, Git hosting services that employ source code identification, some existing source identification tools, as well as the status of open source popularity nowadays.

2.1 Source code control from the Linux kernel: git

Source code control tools that enable code sharing and history tracking are immensely important for developers. Search is usually a feature built on top of these systems. Git [17] is a tool that was born out of the Linux kernel's maintainers frustration with current source code management tools in 2005 [1]. It is a system that allows developers to track changes to their code and share it with others in a decentralised manner. It was inspired by the BitKeeper [8] and Monotone [35] Source Control Systems. Nowadays Git is really popular and there exist many hosting providers for it, although one can always run their own server. Internally, everything is represented as a directed acyclic graph and has the concept of blobs: binary large objects, refs: references, commits: which are a snapshot of the status of the repository of code.

2.2 Compilers and text processing techniques

Compilers are text processing tools. Their main objective, when given some text as input, is to pass it through a well-established pipeline to be able to perform a certain task such as generating code, running it on the fly, or other purposes.

The main steps that can be found in a compiler are:

- **Lexing**: Splitting the input in tokens.
- **Parsing**: Taking the tokens and generating the abstract syntax tree (AST).
- **Code generation**: The AST is walked and code is generated.
- **Optimisations**: Optional step in which the generated code is optimised. That could be removing dead code (code which is never executed), simplifying math and logic expressions, reordering code to make it more performant, using vectorised operations, inlining functions, unrolling loops, etc.

Some of these steps are in occasion merged, but it is considered a good software engineering practice to keep them separated. That reduces coupling which makes easier to add features without introducing breaking changes. It also increases the ease of testing. Semantic analysis is another important stage. It consists of checking things like that the variables or methods being called exist in the current scope. Depending on if this is an interpreter or a compiler, that is done in a different stage.

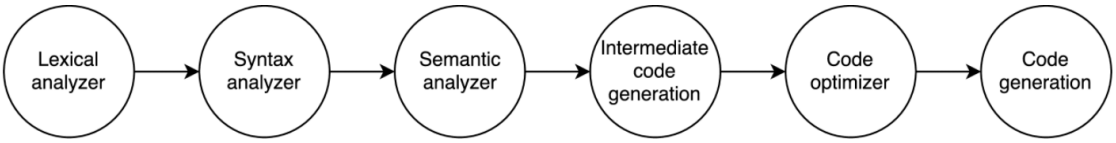

Figure 1: Typical compiler stages

Parsers were trending in research in the seventies, while right now the research area that seems to be pretty active is code optimisations. In the following sections, we will describe these stages a bit more in-depth.

2.3 Lexical analysers

Lexical analysers, also called lexers or scanners, are the tools that convert the input text and split it into different atomic tokens. This is a task that can be performed using a context-sensitive grammar, so a regular language will be powerful enough to identify it. They can be efficiently recognised with deterministic finite automata. Although it is a foundational step of the text processing and compilers pipeline, it is usually a not very complex task.

They are usually implemented by using regular expressions (for example Python's undocumented Scanner class [9]), or by handwriting the states and the transitions of the finite state machine (FSM). Some popular tools which accept the different regular expressions and generate the code for the Lexer are lex [49] and Ragel [41]. For example, Python's lexer is handwritten [10].

2.4 Parsers

Parsing consists in taking the tokens emitted by the Lexer and find how they could fit its grammar. In case there is no way to perform that task, we would have a parsing error. We can generate some data structure, like a tree, that represents how the different parsing rules are applied. That tree is called AST, or Abstract Syntax Tree. It holds all the necessary information to do useful things with it afterwards, like generating machine code or evaluating it directly, as many interpreters do (this technique is called syntax-directed parsing). One of the classic books on compilers is the "Dragon Book" [3], which covers compiler theory as a whole, but focuses on the parsing stage. One important task that parsers usually deal with are precedence rules. For example mathematical expressions' precedences, but not limited to them. Precedence rules have to be designed and applied. Some of the techniques that can be used for taking into account the precedence rules are grammar rewriting, and the operator precedence parsing technique.

The associate grammars that can encode parsers are called context-free grammars. They can be implemented with a myriad of different approaches. One of the easiest to implement by hand, which is really popular, is the so called recursive descent parser which is usually implemented with a function per each non-terminal from the grammar. There are bottom-up approaches, table-driven instead of using recursion, etc. Usually, the latter ones are implemented by means of parser generators. These programs accept a representation of the grammar and some snippets of code to be executed when some derivation rule is matched and generate a parser for that language. They are really handy as they make building and maintaining parsers easier, although some people prefer to handcraft them, since there are some edge cases which are more complex to get right with a parser generator. Most popular languages have available parser generators, and one of the most popular ones, for the C language, is GNU's Bison [25].

Computational complexity depends on variables such as the parsing algorithm and the input grammar. For example, the Cocke–Younger–Kasami (CYK) parsing algorithm, $O(n^3 \cdot |G|)$ in which n is the length of the string to be parsed and $|G|$ is the size of the grammar in Chomsky's Normal Form.

2.5 Code evaluation

In the case of interpreters, the generated code or directly traversed AST has to be evaluated. For example, an AST evaluation starts by the root and uses the appropriate child of every node, following it and applying the required logic: an "if" AST node over a variable, this variable will be evaluated and will check if the value is truthy. If that is the case, one of the children is going to be executed. Otherwise, the other child would be executed, and so on. This part requires to have some memory available to store the variable's value in the current scope, that is done using a symbol table. Symbol tables are used for many things, including as a reference to the variables and functions available in the current scope. That being said, there are several approaches for this and each language's semantics require different implementations.

Other languages code evaluation routines are implemented by means of a virtual machine which executes some sort of intermediate bytecode that has been generated in the code generation phase.

For the following Python code that iterates over a list, built by splitting a string, and then prints each of its elements:

```
for fruit in "apple banana kiwi".split():
    print(fruit)
```

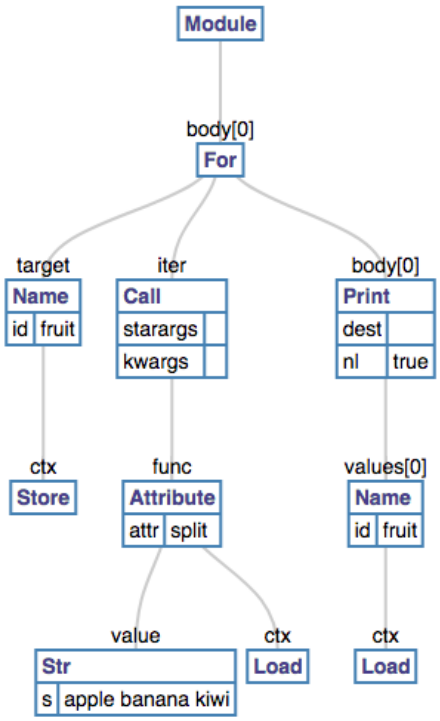

Figure 2: Python AST for the code fragment above from [40]

2.6 Programming languages syntax

At the moment, Linguist [23] supports 469 languages [32] out of the many programming languages that exist. When a programming language is being created, one of the crucial steps of its design is its syntax. After all, it is going to be the first thing developers see and what they read and write every day. The way of expressing ideas varies, but the computing power in Turing complete languages is equivalent. That being said, the Lexing and Parsing phases of the compiler / interpreter are highly influenced by the syntax and grammar of the language, as the Lexer and Parser are pretty foundational parts.

Different programming languages families often share similar syntaxes. For example, functional programming languages like Haskell, Coq, and Erlang have somewhat similar syntaxes.

LISP flavours, like Racket, Common LISP, or Clojure all share a big usage of parenthesis, as S-Expressions are defined using them, while Java, C, C++, use lots of dots for method calls, pointer dereferencing, etc, as well as having a trailing semicolon after each statement.

As we will describe afterwards, this difference in the number of occurrences for a specific token in the code, depending on the programming language used, is one of the core ideas of this work.

2.7 Open source nowadays

As we mentioned before, open source popularity is growing [12]. There are many individuals and companies that directly contribute and benefit to it. In the past years, there have been a multitude of studies that show its growth. Some of the common reasons why corporations are switching to open source software are:

- **Better quality and reliability**: The biggest open source project are used by big companies at a very big scale, in which errors and bugs are more likely to surface. For example, MySQL is used by Google [26] and Facebook [15]. Python is used by these two as well as Dropbox. Of course, they have way more big industrial users, but these three are some of the biggest. Thanks to that, other individuals and companies can benefit from the work that the whole user base does. The patches applied to these code bases are usually peer-reviewed by several members who may have different approaches to software development and may focus on different aspects, like performance, code readability, or testing, which greatly improves the overall quality.
- **More transparency**: Having the source code, means it can be audited, studied, etc. It opens the possibility to treat more software as a white box.
- **Lower cost**: Open Source software and software which is free as in beer is not necessarily the same, but there is a big correlation. That is one of the reasons why many people confuse Open Source Software with software which is free as in beer.

Some authors state that the growth of open source has been exponential [13]. More open source code available means that tools to identify which programming language is used in each file have to be more efficient than before to do analysis over large amounts of repositories.

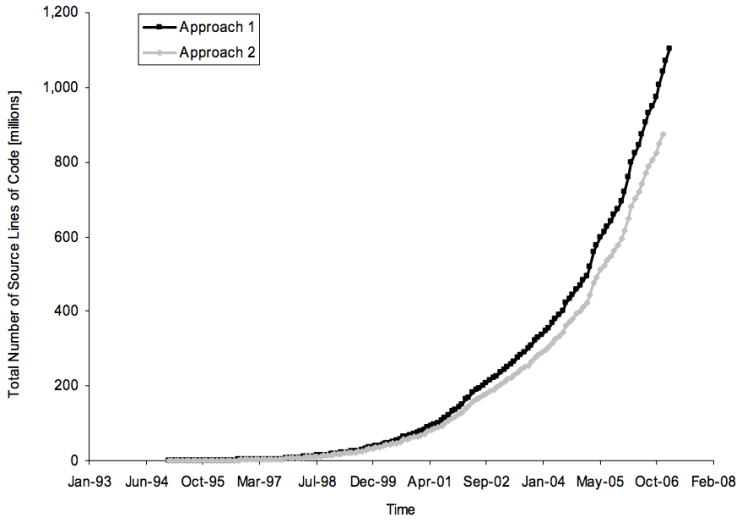

Figure 3: Open source software growth in lines of code over time[13]

Some other papers have examined the benefits (and drawbacks) of open source software more in-depth, like [37]. As this paper mentions, there are clear advantages from the technical standpoint in reliability, security, quality, performance, flexibility of use, developer and tester base, compatibility, and harmonisation. From the business point of view, some of the advantages are: their usually low cost, flexibility by licenses, that it helps to escape vendor lock-in, usually increases collaboration, encourages innovation, offers extra business functionality, and implement defacto standards. They found some drawbacks mainly related to the possible lack of OSS awareness, lack of documentation, sometimes fewer enterprise features, and lack of road-map from a technical point of view. Some of the business faults they found were lack of support (which in my opinion many companies based on OSS are trying to change), lack of marketing, and difficulty on finding appropriate staff, among others.

Using GitHub's [19] publicly released data [18], some people have built visualisations on top of it, like the amount of unique commiters (authors of commits) on GitHub [4]:

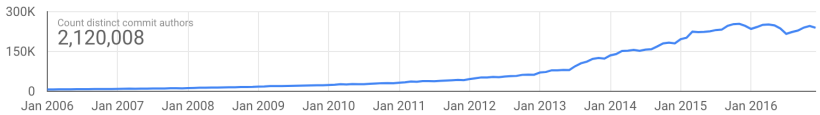

Figure 4: GitHub unique commit authors per over time[4]

This graph is highly relevant as GitHub is one of the biggest and more popular Git hosting platforms nowadays, with many big industry players and individuals using it in a daily basis.

2.8 Popular source code identification tools

Most UNIX-like distributions offer `file(1)` [16], a command that tries to derive information on files, including their programming language if they are text files. GitHub [19] and GitLab [48], two popular Git repository hosting solutions use Linguist and have different ways in which source code identification is used in their user interface.

2.8.1 `file(1)`

`file(1)` [16] is a tool which is available in most UNIX-like systems. It provides information on the provided files including if it is a binary (with, or without symbols), if it is an executable or not, or which kind of file it thinks it is, including the programming language it was written in. To be able to do that it uses "magic files", which are heuristics based on text matching. These magic files are loaded at runtime, and we can see which ones our binary is using by running `$ file --version` which yield `/usr/share/file/magic` on my machine. With `--magic-file` we can pass selected magic files. Magic files contain regular expressions that match some unique characteristic of the language to a type of file or mime-type.

2.8.2 Linguist

Linguist is an open source language identified which started at GitHub and has hundreds of contributors and it is being used in production by several industrial users such as popular Git hosting platforms.

In order to classify a given source code it runs the following strategies in order until one of them matches:

1. **Modeline** checks text editor traces.
2. **Filename** looks for common standardised names, such as Makefile, which is used for the UNIX `make` utility.
3. **Shebang** has a look at the shebang of the file as it specifies the binary used to run the program. The shebang is a header comment in the form of `#!path_to_binary` that indicates the Operating System which binary can run this file.
4. **Extension** checks the extension name and returns success when it is non-ambiguous.
5. **Heuristics** looks for some language-specific keywords that can fully disambiguate the choices.

6. **Classifier** runs a naive-Bayes classifier which was trained using several code samples from various open source projects.

2.9 Source code identification in the User Interface (UI)

As described above, GitHub [19] and GitLab [48] both use Linguist to detect programming languages. In this section, we describe how this is used to build features in their user interface to drill down by them in searches, or gather statistics.

2.9.1 GitHub

GitHub provides a colour bar in each repository as it is shown in 5. Each colour represents a different programming language. Its size in the bar represents how common that language is in this repository. The legend and the numeric percentages are toggled once the bar is clicked. Clicking on one of the toggled programming languages shows all the files written in that language for the repository.

Figure 5: GitHub's toggled language statistics for Linguist [23]

When searching inside of a repository, it allows to drill down by a programming language. Clicking on one of these will make the appropriate filtering. This is immensely useful when looking for specific names on just one or a small number of programming languages in repositories with several languages.

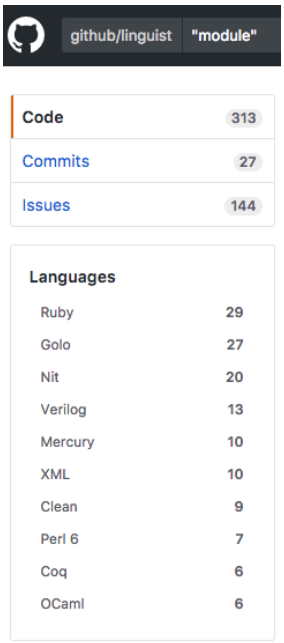

Figure 6: GitHub's search drill down by programming language when searching for the word "module" [22]

We can also filter by the most common programming language inside of an organisation's repositories:

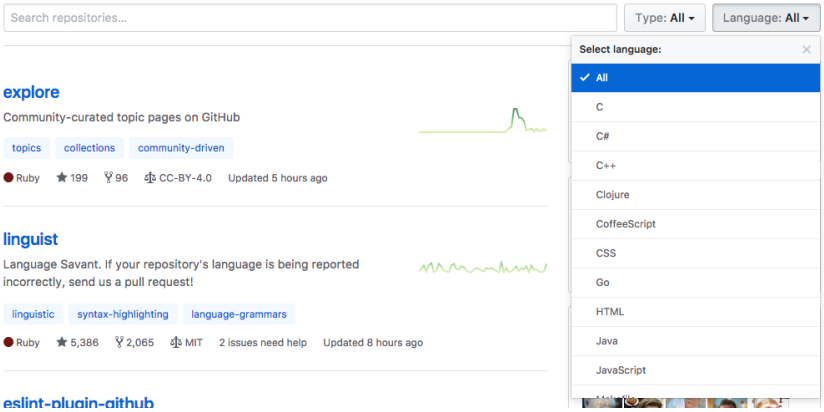

Figure 7: GitHub's organisation filtering by repositories by their main language [21]

Last but not least, rendered file on GitHub's user interface, with syntax highlighting.

Figure 8: GitHub's syntax highlighted Ruby source code rendering [20]

2.9.2 GitLab

GitLab is a GitHub competitor which is open source. It offers a free of charge version called Community Edition. They have fewer features leveraging source code identification, but the ones they employ are for statistics of programming languages in a repository and for syntax highlighting.

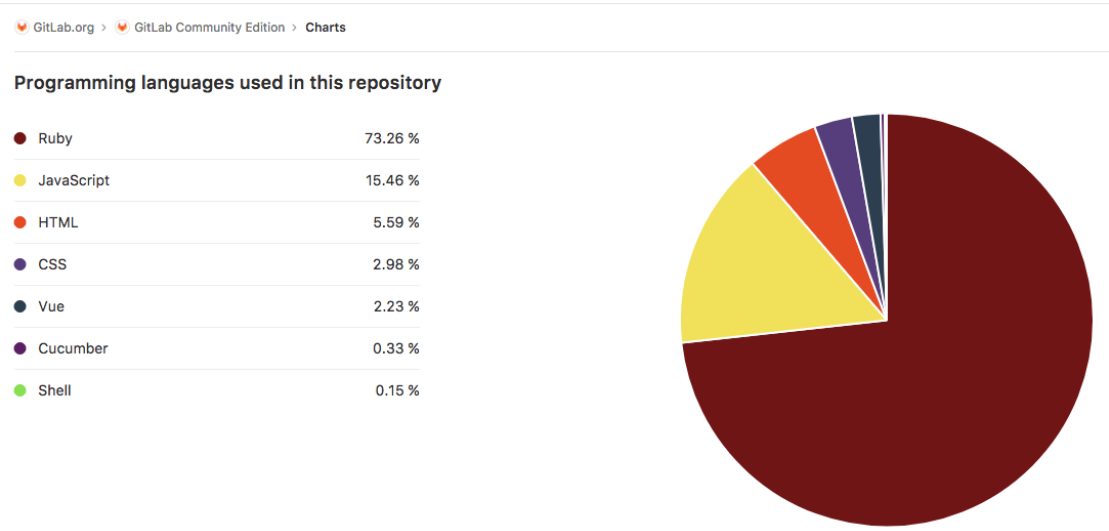

Figure 9: GitLab programming language statistic for Gitlab Community Edition [24]

3 Development Environment

This chapter describes the development environment. We also indicate the libraries, the programming languages, and the hardware platform in which the experiments and benchmarks were run, with their respective versions.

3.1 Third party libraries

We chose Python 3.3 for the model training and visualisation because of its versatility and ease of use. It's well supported and there exist many high-quality machine learning libraries for this language. The third party libraries we used are:

- **scikit-learn** [39] (v0.19.1) is one of the most important libraries we used. It allows rapid prototyping of machine learning models, has a vibrant community, is extensively tested, and includes many interesting utilities.
- **numpy** [52] (v1.13.3) is a transitive dependency of scikit-learn, but we mention it because it's extremely important. Among other features, it offers some array data structures which are used by all our models and that are more efficient than Python's built-in lists.
- **matplotlib** [29] (v2.1.0) was used to generate various graphs.

Pip (v9.0.1) was used to install these libraries.

All of these libraries are really stable and mature, provide nice APIs, and have a huge community that makes troubleshooting problems and looking for ideas way easier. They are also relatively fast, since some of them are implemented as native extensions with especial in-memory data structures that are more efficient than Python's built-ins as they were designed to take advantage of modern CPU's caches. An example of these structures are Numpy arrays.

The tests we did with GitHub's Linguist library were performed in the same hardware. We used Ruby v2.4.0, which we installed with RubyGems and Bundler.

We also used the GNU's C compiler, GCC, as well as LLVM's Clang, for the compilation of the native extensions.

3.2 Our implementation

Our code leverages all the libraries mentioned above. It is written to train several classifiers, check how do they perform, and store these trained models and graphs to disk.

Our repository has the following structure:

- **requirements.txt** lists our Python3 dependencies to be installed.
- **config.py** includes some basic information such as where the Rosetta Code repository is located, where to save the models and graphs, which languages we want to use for training, which are the single character tokens we want to take into account, and which is the file where the multi-character tokens are stored.
- **feature_extraction.py** contains some helper functions to read the input files and calculate how many of the single or multiple character tokens do they have.
- **train.py** has all the necessary helpers for training. With a list of classifiers, the languages we are interested in, and the feature extractor, the `benchmark` function performs the model generation. It also displays the accuracy of the model and calls other helper functions to generate the graphs and save the models to disk.
- **plot_classifiers.py** contains helpers for plotting our models, such as the generated tree in some of our tree-based classifiers.
- **classify.py** classifies a passed file using the selected previously trained model.
- **utils.py** has helpers for benchmarking, such as calculating what is the elapsed time of some block of code.
- **train_benchmark.py** was used to perform benchmarks with the different classifiers. It just invokes the training function from `train.py`.
- **script/** contains shell scripts to set up the development environment.
- **data/** is where the multi-character tokens and a preprocessing script live.
- **output/** is where the trained models and corresponding graphs are stored by default.

All the code was abstracted as much as possible to have a generic interface. It receives a list of classifiers with their parameters and outputs some information about the training data, such as if we have enough samples from each language or not, and the elapsed time to train each classifier, as well as their accuracy.

3.2.1 Dependency graph

We have plotted below the dependency graph of the project's main files. This directed graph indicates which of the project's files are required for each of the source code files. The graph is useful to develop a mental model on how the code is structured, which modules call which modules, and it enables an easier refactoring experience.

In text format:

- **config.py** requires no other file
- **plot_classifiers.py** requires `config.py`
- **classify.py** requires `config.py` and requires `feature_extraction.py`
- **feature_extraction.py** requires `config.py`
- **train.py** requires `config.py`, `feature_extraction.py`, and `utils.py`. This is the core file which invokes routines form others files and has the main logic of the training and testing procedures.
- **train_benchmark.py** requires `benchmark.py`
- **utils.py** requires no other file

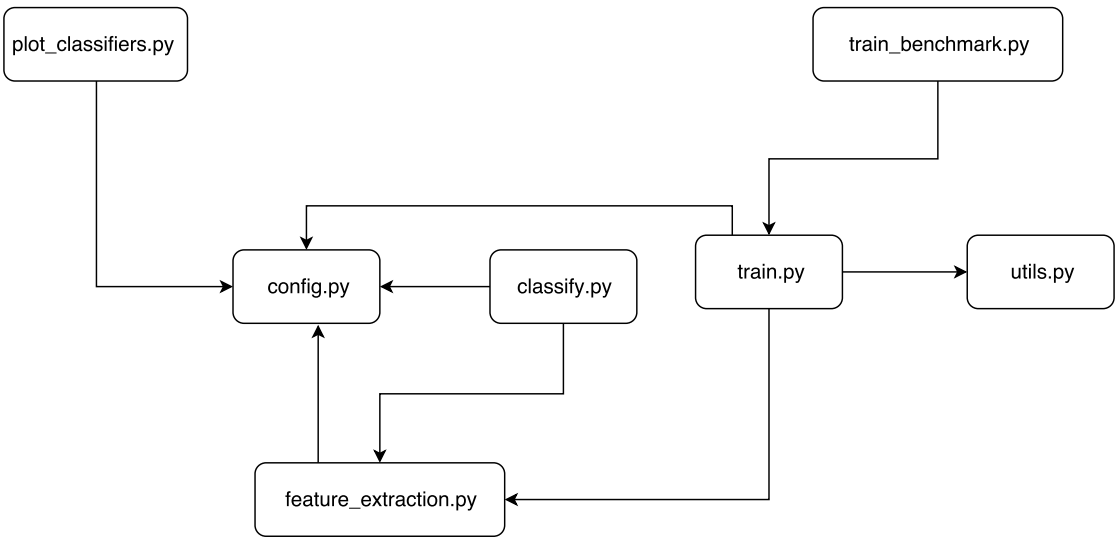

Figure 10: Dependency graph of the project's main files

The following configuration was used to perform an initial benchmarking of classifiers with some default parameters.

This code defines which classifiers are going to be used, fetches the languages variable from the configuration (which includes the languages that we are going to use for training) and the number of samples we are going to get from Rosetta Code. It finally passes everything to the `benchmark` function, which trains the passed classifiers, generates the trained models and confusion matrices, tests the accuracy of the classifier, and finally exits.

```
from train import *
from config import LANGUAGES

SAMPLES_AMOUNT = 1000

classifiers = [
    MultinomialNB(),
    KNeighborsClassifier(
        len(LANGUAGES),
    ),
    DecisionTreeClassifier(
        max_depth=100,
    ),
    RandomForestClassifier(
        max_depth=100,
        n_estimators=len(CHAR_FEATURES),
        max_features=len(CHAR_FEATURES),
    ),
    AdaBoostClassifier(),
    ExtraTreesClassifier(
        n_estimators=len(CHAR_FEATURES),
        max_features=len(CHAR_FEATURES),
        n_jobs=4,
        random_state=0,
    ),
]

benchmark(
    LANGUAGES,
    SAMPLES_AMOUNT,
    classifiers,
    extract_token_features,
)
```

The output of running the snippet of code above shows the preprocessing time, the languages that were used, the feature extractor, etc. It also shows a warning on class imbalances and the training time of each classifier. Finally, it saves the model and the confusion matrix of each classifier.

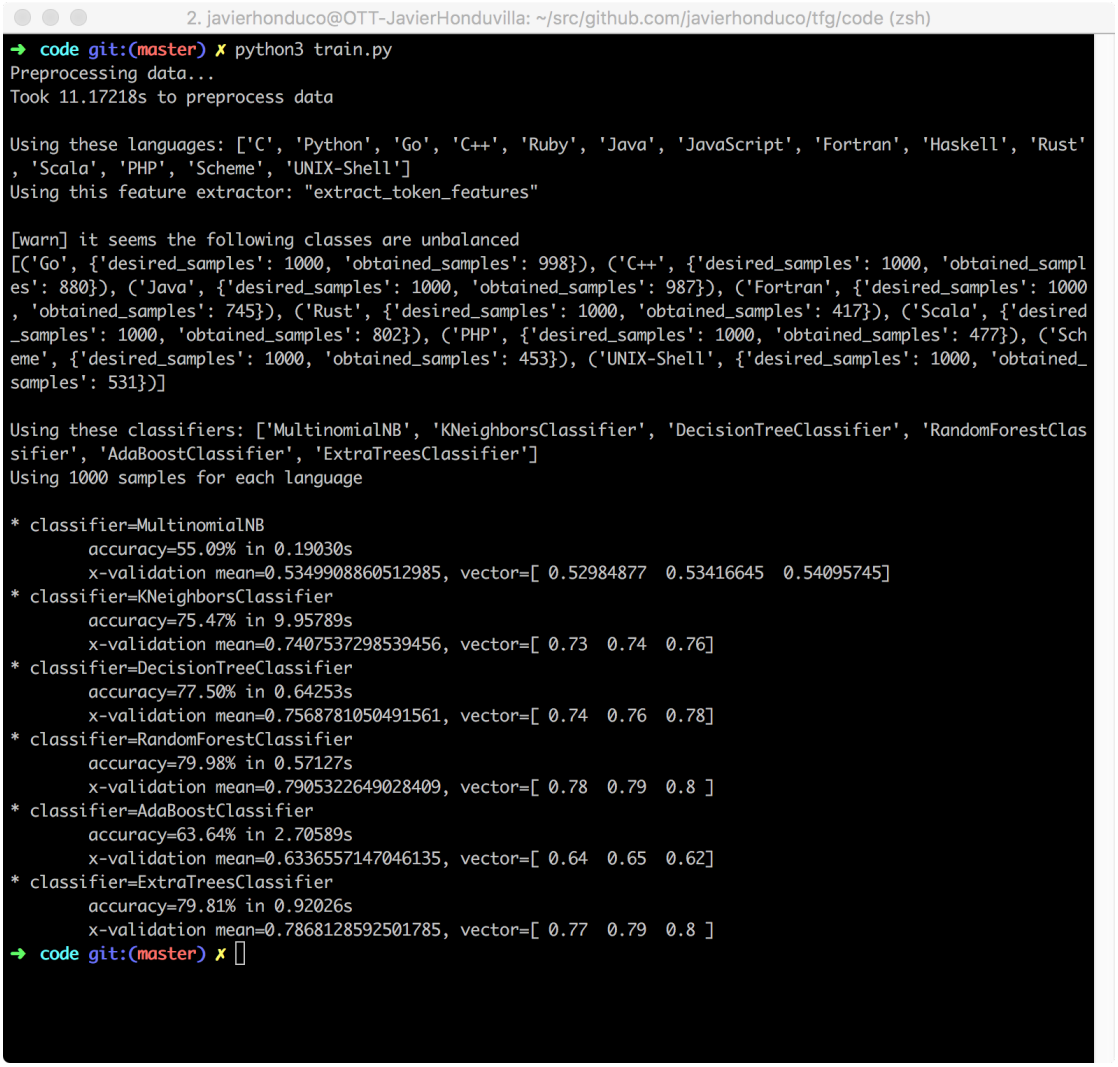

Figure 11: Screenshot of the terminal output of the running the model training script with the configuration above, showing the classifiers used, the accuracy of each model and the cross-validation mean vector from Scikit learn.

3.3 Developer tools and other software

Some of the tools we used to develop the code and experiments of this bachelor thesis are:

3.3.1 iTerm2

iTerm2 [30] is a popular terminal alternative for macOS. Some of the useful features it offers are the ability to split the terminal window into panes and select one using keyboard shortcuts, as well as the ability to select how much terminal history it should buffer. Besides these useful features, iTerm2 also has a really good stability and performance.

3.3.2 Vim

Vim [51] is a popular text editor in UNIX systems. Its most prominent feature is the high degree of customisation it allows. There are plugins written for pretty much every programming language and frameworks. Its memory usage and performance, especially when dealing with large files is another of its major selling points.

3.3.3 Git and GitHub

We checked our source code in a Git repository hosted on GitHub. This allowed to share and review code more easily and rollback changes when regressions were introduced.

3.3.4 Operating system

The development machine runs macOS v10.12.6 code-named “Sierra”.

3.4 Hardware

For the benchmarks, I used my personal laptop: a MacBook Air mid-2012 (Intel Core I5 1.8 GHz, 8 Gb RAM 1600 MHz DDR3).

4 Project development

In this section we describe how this project has been developed, what are the proposed objectives, which development methodology we have used and in which phases we decided to split it up. We discuss the use cases, the functional and non-functional requirements, and finally we describe the current socio-economic environment in which, among other things, we show the broken down budget.

4.1 Objectives

Our main objective is to develop a system that could enable source code identification in an accurate and efficient manner. During the first meeting, we discussed several possible implementation ideas, such as using machine learning, grammar techniques, etc. We searched for already existing approaches to this problem and found Linguist [23]. We also made sure this is a relevant problem to try to solve and bumped into Linguist usage by some of the major Git [17] hosting companies.

We ended up choosing to experiment with machine learning models and text classification techniques as it seemed a good approach for source code identification.

4.2 Development methodology and phases

For the exploratory stage of this project, we started by implementing a hacked together naïve-Bayes classifier that was trained with one-character tokens.

Once we decided we wanted to go for a machine learning and text processing approach, we started researching machine learning libraries for Python. We ended up choosing scikit-learn [39] and implemented a basic prototype based on the previously handcrafted classifier. This version built a vector with the occurrences of those one-character tokens and fed it into scikit-learn's naïve-Bayes model.

As we wanted to try several classifiers and tune their parameters, we coded some helpers to help us do benchmarks, letting us know the time that it took to complete each training and the accuracy of each model while performing cross-validation. We also built other helpers that are in charge of rendering and saving the confusion matrices to disk, as well as the models.

Once we iterated a couple of times on what was our objective and what our approach would be, plus some experiments based on the initial ideas, we started developing the final code.

4.2.1 Gantt chart

In this section we present a simplified Gantt chart that reflects the planning of our work:

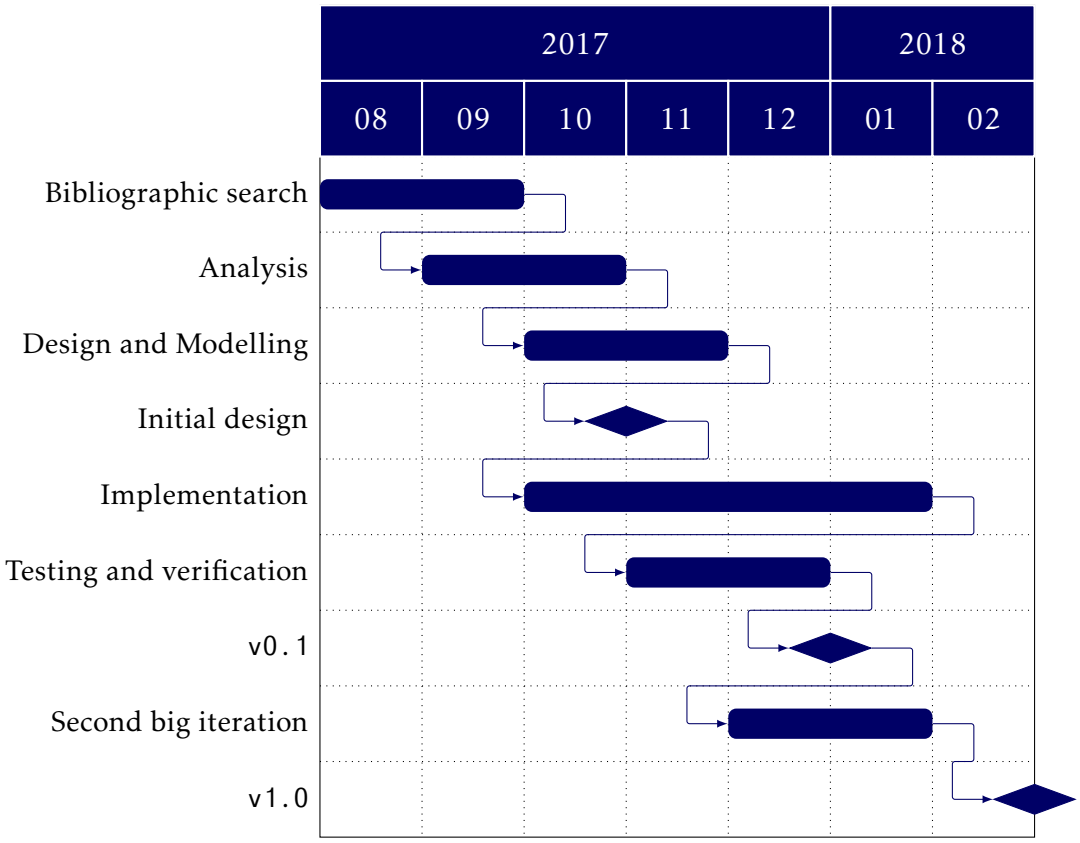

Figure 12: Gantt chart of the project

4.3 Requirements and use case analysis

In this section we describe the different requirements and use case analysis of the tool we develop. This is important so we can have a clear contract with the different stakeholders of the project and make sure that we are reaching our project goals with respect to features, timing, or any other requirements.

4.3.1 Requirements

The attributes are specified using this table:

Requirement type			
ID	NF/F-Integer	**Stakeholder**	Person interested in this feature
Name	Name of the requirement		
Description	Description of the requirement		
Relations	Relation with other requirements, if any		
Importance	High, Medium, or Low	**Priority**	High, Medium, or Low

Table 1: Requirements table template

The description of each attribute:

- **ID**: Uniquely identifies each requirement. The NF prefix stands for non-functional and F for functional. It is followed by a dash and a numeric identifier with as many zeroes as needed to left pad.
- **Stakeholder**: Is the person interested in this feature. In some cases, it may refer to the end user of the library with already generated models or to the developer that wants to train some classifier tuning parameters or adding more languages, for example.
- **Name of the requirement**: Identifier that gives an at a glance short description of what the requirement is about.
- **Description of the requirement**: More in-depth description of what that requirement is about.
- **Relations**: Which requirements are linked with this one, if any.
- **Importance**: How important is this feature for the stakeholder. Can be one of the following:
  - High: This feature has to be implemented.
  - Medium: This feature is interesting to have, but not enforced.
  - Low: A nice extra, but not necessary.
- **Priority**: How we are prioritising this requirement. Can be:

- High: This requirement must be implemented early in the development process.
- Medium: This requirement must be implemented after the high priority tasks have been finished.
- Low: This requirement must be implemented once the medium priority classes have been finished.

4.3.2 Functional system requirements

Functional system requirement			
ID	FR-01	**Stakeholder**	Developer using the library to train models
Name	Generation of the the trained models		
Description	The program must accept a list of the names of the programming languages we are interested in and generate the training models in the appropriate folder		
Relations			
Importance	High	**Priority**	High

Table 2: Functional requirement FR-01. Generation of the trained models.

Functional system requirement			
ID	FR-02	**Stakeholder**	Developer using the library to train models
Name	Generation of the confusion matrices		
Description	The program must accept a list of the names of the programming languages we are interested in and generate the confusion matrices in the appropriate folder		
Relations			
Importance	High	**Priority**	High

Table 3: Functional requirement FR-02. Generation of the confusion matrices.

Functional system requirement			
ID	FR-03	**Stakeholder**	Developer using the library to train models
Name	Use cross validation		
Description	The training procedure must perform cross validation: receive part of the training set and use the rest of the input for testing the performance of the classifier.		
Relations			
Importance	High	**Priority**	High

Table 4: Functional requirement FR-03. Usage of cross validation.

Functional system requirement			
ID	FR-04	**Stakeholder**	Developer using the library to train models
Name	Output the accuracy of the training		
Description	At the end of the training procedure, the program must display how accurate is each model		
Relations	FR-03		
Importance	High	**Priority**	High

Table 5: Functional requirement FR-04. Output the accuracy of the training.

Functional system requirement			
ID	FR-05	**Stakeholder**	Developer using the library to train models
Name	Needs MacOS "Sierra" or higher installed		
Description	The computer in which the software is running must run MacOS "Sierra" or higher		
Relations			
Importance	Medium	**Priority**	Medium

Table 6: Functional requirement FR-05. OS requirements.

Functional system requirement			
ID	FR-06	**Stakeholder**	Developer using the library to train models
Name	Homebrew[28] must be installed		
Description	The computer running MacOS Sierra or higher must have Homebrew installed		
Relations	FR-04		
Importance	Medium	**Priority**	Medium

Table 7: Functional requirement FR-06. Homebrew must be installed.

Functional system requirement			
ID	FR-07	**Stakeholder**	Developer using the library to train models
Name	Must have Python3 installed		
Description	This software has been designed and developed to run on Python3, hence any version of Python3 must be installed		
Relations	FR-05		
Importance	High	**Priority**	High

Table 8: Functional requirement FR-07. Python 3 must be installed.

Functional system requirement			
ID	FR-08	**Stakeholder**	Developer using the library to train models
Name	Must have git[17] installed		
Description	Our source code and the Rosetta Code source repositories employ Git as a SCM		
Relations	FR-05		
Importance	High	**Priority**	High

Table 9: Functional requirement FR-08. Git must be installed.

4.3.3 Non functional system requirements

Non functional system requirement			
ID	NFR-01	**Stakeholder**	Developer using the library to train models
Name	Ease of updating the Rosetta Code repository[44]		
Description	Updating the Rosetta Code repository must be an easy task		
Relations	FR-07		
Importance	High	**Priority**	Medium

Table 10: Non-functional requirement NFR-01. Updating the Rosetta Code repository must be easy.

Non functional system requirement			
ID	NFR-02	**Stakeholder**	Developer that uses the library for training or classification
Name	Most of its configuration must be extracted		
Description	Most of the configuration for the project must be extracted to a configuration file so tuning the project's parameters is easier		
Relations			
Importance	Medium	**Priority**	Low

Table 11: Non-functional requirement NFR-02. Most of the project's configuration must be extracted.

Non functional system requirement			
ID	NFR-02	**Stakeholder**	Developer
Name	Trains more than 10 languages in less than 2 minutes		
Description	The classifier must be able to train more than 10 languages in less than 2 minutes of CPU Wall time.		
Relations			
Importance	High	**Priority**	High

Table 12: Non-functional requirement NFR-03. Trains the model with a required amount of languages in certain time.

Non functional system requirement			
ID	NFR-04	**Stakeholder**	Developer
Name	Trains more than 10 languages and produces a model of reasonable size		
Description	Trains more than 10 languages and produces a model that is less than 100MB in size		
Relations			
Importance	Medium	**Priority**	High

Table 13: Non-functional requirement NFR-04. The output models must have a reasonably low size.

Non functional system requirement			
ID	NFR-05	**Stakeholder**	Developer
Name	Trains more than 10 languages and has an accuracy bigger than 70%		
Description	Trains more than 10 languages and has an accuracy bigger than 70%.		
Relations			
Importance	High	**Priority**	Medium

Table 14: Non-functional requirement NFR-05. Definition of mininimum required accuracy.

Non functional system requirement			
ID	NFR-06	**Stakeholder**	Developer using the library for model training
Name	Performs the identification of a sample source text file in less than 1 second		
Description	Performs the identification of a sample source text file in less than 1 second of CPU Wall time		
Relations			
Importance	High	**Priority**	High

Table 15: Non-functional requirement NFR-06. The identification of one sample of source code must be done under a second.

Non functional system requirement			
ID	NFR-07	**Stakeholder**	Developer using the library for model training
Name	Minimum samples for each language		
Description	Each language must have, at least, 400 sample source code files		
Relations			
Importance	High	**Priority**	High

Table 16: Non-functional requirement NFR-07. Defines the number of minimim required samples.

Non functional system requirement			
ID	NF-08	**Stakeholder**	Developer using the library for model training
Name	Must have an easy installation of 3rd party libraries		
Description	Installing the required 3rd parties libraries should be an easy process involving just one command		
Relations			
Importance	High	**Priority**	High

Table 17: Non-functional requirement NFR-08. Must have an easy installation of 3rd party librarie.

Non functional system requirement			
ID	NF-09	**Stakeholder**	Developer using the library for model training
Name	Must have a simple API that allows easy extension		
Description	The provided API to train several classifiers should be simple an intuitive		
Relations			
Importance	High	**Priority**	High

Table 18: Non-functional requirement NFR-09. Must have a simple API that allows easy extension

4.3.4 Use cases

The use cases show the possible ways in which users are going to interact with this project. The layout of a typical use case looks like this:

Use case	
ID	Numeric identifier
Name	The name of the use case
Stakeholders	Who are the stakeholders interested in the use case
Precondition	What are the necessary conditions for this use case to take place
Effect	What is the effect of applying the use case

Table 19: Use cases template

The description of each attribute:

- **ID**: Uniquely identifies each use case. It is followed by a dash and an integer with as many zeroes as needed in the left pad.
- **Name of the use case**: Identifier that gives an at a glance short description of what the use case is about.
- **Stakeholders**: The people for which this use case is intended
- **Precondition**: Necessary conditions for this use case to take effect.
- **Effect**: Changes that the application of this use case will produce.

Use case	
ID	UC-01
Name	Install dependencies
Stakeholders	Developer that wants to train some model using this software
Precondition	The software developed in this bachelor thesis must be downloaded, as well as Rosetta Code's, a Python environment, and an internet connection
Effect	Dependencies are installed

Table 20: Use case UC-01. Dependency installation.

Use case	
ID	UC-02
Name	Use default models
Stakeholders	Developer that wants to try this software
Precondition	Dependencies must be installed
Effect	The developer will be able to try the pre-built classifier

Table 21: Use case UC-02. Usage of default models.

Use case	
ID	UC-03
Name	Perform a model training with all the defaults
Stakeholders	Developer that wants to try this software
Precondition	Dependencies must be installed
Effect	The developer will be able to reproduce the pre-built classifier

Table 22: Use case UC-03. Model training with default parameters.

Use case	
ID	UC-04
Name	Add a new programming language to the model
Stakeholders	Developer that wants to try this software
Precondition	Dependencies must be installed, add the language to the config
Effect	The developer will be able to generate a model with the specified programming languages

Table 23: Use case UC-04. Addition of a new programming language to a model.

Use case	
ID	UC-05
Name	Modify configuration
Stakeholders	Developer that wants to try this software
Precondition	Dependencies must be installed, configuration must me modified at will
Effect	The developer will be able to generate a model with the specified configuration

Table 24: Use case UC-05. Configuration modification.

Use case	
ID	UC-06
Name	Tune models parameters
Stakeholders	Developer that wants to try this software
Precondition	Dependencies must be installed, parameters passed to a classifier must be tuned
Effect	The developer will be able to generate a model with some defined parameters

Table 25: Use case UC-06. Tuning of model parameters.

Use case	
ID	UC-07
Name	Obtain results
Stakeholders	Developer that wants to try this software
Precondition	Dependencies must be installed, training script must be run
Effect	The developer will obtain training models and confusion matrices

Table 26: Use case UC-07. Production of result.

4.4 Socio-economic environment

In this section we describe the budget planned for this project. We define the estimated amount of time and the number of developers that would be needed to develop a project like this. We also quantify the cost of the used hardware and we finally describe the socio-economic impact of this work.

4.4.1 Budget

This section breaks down the costs by hardware and estimated salaries. We have estimated that we would need 4 months of part-time (20h/week) work of a software developer and 2 months worth of a data scientist working 20h/week as well. The only hardware used for this project has been a MacBook Air for the software developer and a MacBook Pro for the data scientist which are described in the "hardware" section.

		Price in EUR
Hardware	**MacBook Air**	1.300
	MacBook Pro 15 inches	2.800
Salaries	**Software developer**	15EUR/h*320h=4.800
	Data scientist	25EUR/h*160h=4.000
Total		12.900

Table 27: Estimated budget of the project in EUR

This budget estimate doesn't take the taxes into account. In Spain, the VAT is 21%, so the total budget would be of 15.609EUR.

4.4.2 Socio-economic impact

This section assesses the socio-economic impact of the developed work.

The social benefits of this project are really scoped, as it would only affect individuals and companies interested in source code identification. Most of these stakeholders are going to be developers or data scientists, as they are the ones interested in better tooling to do their job more efficiently (source code highlighting, better search, ...), or creating insights from data.

The economic impact would be close to zero, as this is not planned to be directly sold. All this software has been developed as open source under the MIT license [**mit-license**]. We could potentially take economic advantage of this system by offering enterprise support, a set of more advanced features or increased performance in a paid version, or even a hosting service, following what many companies in the open source world do, such as Gitlab [48].

5 Experiments

In this section, we describe the experiments that we have performed to test our hypothesis and see which techniques are the most accurate and efficient.

The training and testing corpus we have used were extracted from the Rosetta Code [44] repository on GitHub [19]. The Rosetta Code project is a collaborative website where people submit solutions to Computer Science problems in different languages, hence its name.

First of all, we have applied cross-validation with a split of 20% for the test data set and the rest 80% was used for training. We used at most 1000 samples per language.

We have chosen 14 languages to do our experiments. They represent a variety of programming paradigms, some of them have been around for a long time, while others started booming just a couple of years ago.

Language	Samples count
C	1078
Python	1452
Go	998
C++	880
Ruby	1104
Java	987
JavaScript	1104
Fortran	745
Haskell	1333
Rust	417
Scala	802
PHP	477
Scheme	453
UNIX-Shell	531

Table 28: The 14 programming languages chosen and the amount of samples we got from the Rosetta Code project

Some of the metrics we are interested in are:

- **Classifier training time**: How long does it take to build a model with our training set.
- **Performance of the classifier**: Throughput of the classifier; how many source code files can it classify per unit of time.
- **Memory usage**: We want to keep it as low as possible. Even though RAM is considered a reasonably cheap resource nowadays, increased RAM usage still produces more elevated cost and operational burden (for example, more GCs can be triggered and the service will have more long-lasting pauses due to GC pressure in stop-the-world implementations).
- **Ease of development**: User experience by different stakeholders
  - By the classifiers developers
  - By users of the library. The installation process, the ease of use of the API, etc.
- **Availability and price** of the underlying required software and hardware necessary to run each classifier, as well as their licenses as that could be a deal breaker for some entities.

5.1 Methodology

In order to design our experiments we chose to go for a well-established approach that we learnt during our machine learning courses: we picked each dimension of the problem and changed some of the parameters of it. Afterwards, we run some benchmarks on it to see how well it performs and how its throughput varies.

One important axis is how are we going to design our feature vectors. We explain it in the next sections, but we chose two different vectors: one representing single character tokens, and the other one representing tokens of one or more characters (from now called multi-character tokens), like the ones we can find in most programming languages reserved words/terminal productions in a grammar.

The classifier's parameters were at first chosen arbitrarily and we run our benchmarks on it to establish some initial metric. We show their confusion matrix as well. As usual, we started tuning the different parameters in increments and writing down the results here. We decided to show the confusion matrix for the one with the highest accuracy.

Confusion matrices are extremely useful in machine learning because they provide an at a glance view of how the classifier is performing. In both axes, we can find all the different classes. Whenever a classifier makes a prediction it adds 1 to the intersection with the other class in the matrix. That means it's going to do so in the diagonal when the prediction is correct and anywhere else when it's not. Hence, an accuracy of a 100% would be represented by only the diagonal being greater than zero. In order to be easier to grasp, the background of each cell has a more intense colour when the number is higher.

Afterwards, we run some further experiments with the ones with the highest performance, like seeing what their classification throughput is and how much memory do they take.

We end up by commenting on the experiment's accuracy and performance.

5.2 Feature vectors

We started by implementing a simple feature extractor in Python. Python was chosen as it's a language we are comfortable with and that has a vibrant machine learning community with lots of high-quality libraries, such as the one we are using for machine learning: Scikit-learn.

The feature extractor builds a feature vector for each language using code samples. Finally, feature vectors were fed into several supervised classifiers. We have used the following classifiers found in Scikit-learn and explained more in-depth in the following section:

- ***K*-nearest Neighbours**
- **Decision Trees**
- **Multinomial Naive Bayes**
- **Random Forest**
- **Ada-Boost**
- **Extra (EXTremely Randomised) Trees**

Even though we have selected the 14 aforementioned programming languages, we have tried to make the code as extensible as possible so other languages could easily be added and more experiments could be made in the future.

For each experiment, we added a table of its accuracy and time that it takes to train each classifier, as well as the confusion matrix for the classifier with the best performance. The other confusion matrices can be found in the appendix, and the trained models can be easily generated with the provided code but are as well provided in the repository.

5.2.1 Classifiers overview

This section gives a short explanation on how the classifiers work under the hood. All these classifiers that we have used can be found in the amazing Scikit-learn Python library.

5.2.1.1 K-nearest Neighbours

This algorithm is one of the simplest machine learning algorithms. It can be used for regression and classification. The way it works is by computing the distance and minimising it from the test vector we want to classify to the samples used for training. When k is 1, the closest one is chosen. Otherwise, the most common one of the k closest training samples are chosen. That makes that depending on the ordering of the training data, it can yield slightly different results.

This classifier can be found in `sklearn.neighbors.KNeighborsClassifier`

5.2.1.2 Decision trees

Decision trees can be used for classification. An algorithm constructs the tree in which non-leaf nodes do attribute testing. For example, checking that a number is bigger than X, and depending if that condition evaluates to true or to false, we should choose the right or the left child, respectively. Leaf nodes correspond to the different classes. To classify some data, we start in the root and we evaluate each node and follow the appropriate child until we reach a leaf node, which will let us know what the predicted label is.

This classifier can be found in `sklearn.tree.DecisionTreeClassifier`

5.2.1.3 Multinomial Naive Bayes

Naive Bayes is a family of algorithms which apply Bayes' theorem. That means that each feature is taken as independent of the others. They are fast, well scalable, and have been outperforming other complex approaches in a variety of tasks. We have decided to use the Multinomial flavour as it's typically used in text classification tasks, as each we are representing the number of times a single token (word) happen in each file (document).

This classifier can be found in `sklearn.naive_bayes.MultinomialNB`

5.2.1.4 Random Forest

Random forests are an ensemble method that can be used for classification and regression. They generate many decision trees with the advantage of not over-fitting as much as the Decision Trees classifiers counterparts when they generate deep trees. They work similarly to Decision Trees classifier, but the class that it's chosen it's the *mode* of the classes.

This classifier can be found in `sklearn.ensemble.RandomForestClassifier`

5.2.1.5 Ada-Boost

Is an ensemble method that takes several weak classifiers and trains them with slightly modified versions of the data. The result is aggregated and the weights are modified depending if they got the prediction right or not.

This classifier can be found in `sklearn.ensemble.AdaBoostClassifier`

5.2.1.6 Extra (EXTremely Randomised) Trees

Similar to the Random Forest algorithm, but with additional randomness. While Random Forest computes the locally optimal combination of feature/split, this algorithm randomises it.

This classifier can be found in `sklearn.ensemble.ExtraTreesClassifier`

5.2.2 Single character tokens

This experiment uses single character tokens as features. The hypothesis is that a small set of common, one character tokens, could be enough to identify the programming language.

The were chosen because in our experience they seemed to be quite popular tokens in many programming languages:

; ({ [: . | $ & < > # ! ? @

5.2.2.1 Results

- **With default parameters**:

	Accuracy (%)	Time (s)
Multinomial Naive Bayes	26.31	**0.01654**
K-Neighbours	63.86	0.37366
Decision Trees	63.51	0.09944
Random Forest	69.40	0.86643
Ada-Boost	46.90	0.64589
Extra Trees	**72.67**	0.30490

Table 29: Single character tokens results

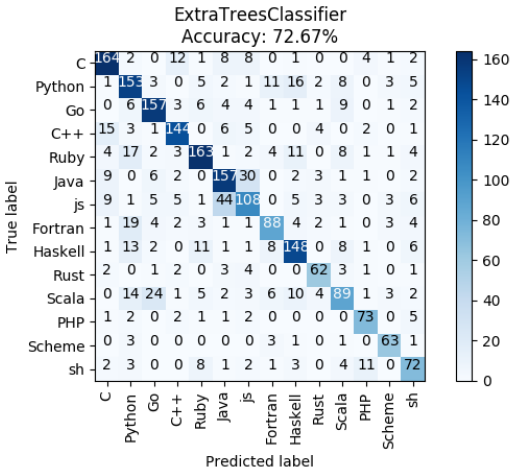

Figure 13: Extra Trees confusion matrix for single character tokens (not normalised) with default parameters

- **Experiments with different parameters**:

  We decided to vary the parameters from 1 to 1000 or 10000 (depending on the classifier maximum allowed number) in increments of x10 to see which configuration would be the most accurate and if it would be reasonable in terms of speed.

  In the tables above, the row with emphasis is the one with the highest accuracy. When 2 rows have the same accuracy, the one with emphasis would be the one with the lowest timing.

  - **K-Neighbours**

K	Accuracy (%)	Time (s)
1	63.42	0.22031
10	**64.84**	**0.33651**
14	63.86	0.36236
100	60.01	0.55934
1000	45.97	1.37451

Table 30: Single character tokens parameter experimentation for K-Neighbours

  - **DecisionTreeClassifier**

max_depth	Accuracy (%)	Time (s)
1	17.09	0.02369
10	**65.06**	**0.07261**
14	64.13	0.10521
100	64.04	0.10060
1000	63.91	0.10174

Table 31: Single character tokens parameter experimentation for DecisionTreeClassifier

  - **RandomForestClassifier**

max_depth	Accuracy (%)	Time (s)
1	17.14	0.11966
10	69.18	0.61773
100	69.80	0.85281
1000	**70.46**	**0.90468**
10000	69.80	0.86805

Table 32: Single character tokens parameter experimentation for RandomForestClassifier varying max_depth with rest of parameters as defaults

n_estimators	Accuracy (%)	Time (s)
1	63.46	0.86805
10	70.02	0.86805
100	71.21	0.86805
1000	**71.30**	**0.86805**
10000	(blew up)	(blew up)

Table 33: Single character tokens parameter experimentation for RandomForestClassifier varying n_estimators with rest of parameters as defaults

n_features	Accuracy (%)	Time (s)
1	71.30	0.16041
5	**71.79**	**0.32810**
10	70.90	0.56322
15	70.68	0.77310

Table 34: Single character tokens parameter experimentation for RandomForestClassifier varying n_features with rest of parameters as defaults

– **ExtraTreesClassifier**

n_estimators	Accuracy (%)	Time (s)
1	60.05	0.15068
10	71.88	0.35714
100	73.65	1.42328
1000	**73.91**	**15.02232**
10000	(blew up)	(blew up)

Table 35: Single character tokens parameter experimentation for ExtraTreesClassifier varying n_estimators with rest of parameters as defaults

max_features	Accuracy (%)	Time (s)
1	68.95	0.39289
5	71.21	0.34462
10	72.50	0.31929
15	**72.67**	**0.30107**

Table 36: Single character tokens parameter experimentation for ExtraTreesClassifier varying max_features with rest of parameters as defaults

Analysis of results: The configuration with the best performance for the single character tokens experiments has been the ExtraTreesClassifier, with an accuracy of 73.91% in 15s using n_estimator=1000 and max_features=14 (number of languages). It was followed really closely, 72.67% accuracy, by the same classifier using 15 as max_features and n_estimators=14 (number of languages) with a way lower timing of 0.3s.

5.2.3 Multi-character tokens

For this experiment, we have gathered all the reserved words from the programming languages we were analysing and we have removed the duplicates.

They can be found in the appendix and in the project repository, as it is too big to fit inline.

5.2.3.1 Results

- **With default parameters**:

	Accuracy (%)	Time (s)
Multinomial Naive Bayes	55.09	**0.15575**
K-Neighbours	75.47	8.55795
Decision Trees	77.06	0.47534
Random Forest	**79.89**	0.53402
Ada-Boost	63.64	2.20398
Extra Trees	79.81	0.73692

Table 37: Multicharacter tokens results

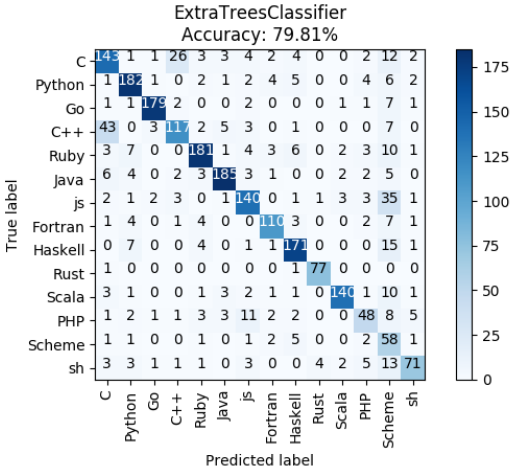

Figure 14: Extra Trees confusion matrix for multiple character tokens (not normalised) with default parameters

- **Experiments with different parameters**:
  - **K-Neighbours**

K	Accuracy (%)	Time (s)
1	73.60	6.42013
10	**75.95**	**8.71088**
14	75.47	8.99971
100	71.26	9.05237
1000	48.27	10.75847

Table 38: Multiple character tokens parameter experimentation for K-Neighbours

  - **DecisionTreeClassifier**

max_depth	Accuracy (%)	Time (s)
1	16.34	0.19711
10	64.04	0.33480
14	77.10	0.46774
100	**77.10**	**0.46356**
1000	76.97	0.45710

Table 39: Multiple character tokens parameter experimentation for DecisionTreeClassifier

  - **RandomForestClassifier**

max_depth	Accuracy (%)	Time (s)
1	41.28	0.21747
10	74.80	0.31949
100	**79.67**	**0.56342**
1000	79.54	0.54650
10000	79.54	0.53995

Table 40: Multiple character tokens parameter experimentation for RandomForestClassifier varying max_depth with rest of parameters as defaults

n_estimators	Accuracy (%)	Time (s)
1	61.38	0.16117
10	75.47	0.28180
100	**78.61**	**1.69656**
1000	78.17	15.92109
10000	78.43	159.96337

Table 41: Multiple character tokens parameter experimentation for RandomForestClassifier varying n_estimators with rest of parameters as defaults

n_features	Accuracy (%)	Time (s)
1	66.87	0.22371
5	74.45	0.25146
10	**77.46**	**0.31354**
15	76.97	0.35075

Table 42: Multiple character tokens parameter experimentation for RandomForestClassifier varying n_features with rest of parameters as defaults

– **ExtraTreesClassifier**

n_estimators	Accuracy (%)	Time (s)
1	66.83	0.25995
10	79.14	0.60375
100	**81.40**	**2.83424**
1000	81.22	31.23288
10000	(blew up)	(blew up)

Table 43: Multiple character tokens parameter experimentation for ExtraTreesClassifier varying n_estimators with rest of parameters as defaults

max_features	Accuracy (%)	Time (s)
1	76.44	0.79127
5	78.61	0.66973
10	78.57	0.64903
15	**79.81**	**0.77102**

Table 44: Multiple character tokens parameter experimentation for ExtraTreesClassifier varying max_features with rest of parameters as defaults

Analysis of results: The configuration with the best performance for the single character tokens experiments has been the ExtraTreesClassifier, with an accuracy of 81.22% in 31s using n_estimator=1000 and max_features=14 (number of languages). It was also followed really closely, 79.81% accuracy, by the same classifier using 15 as max_features and n_estimators=14 (number of languages) with a way lower timing of 0.8s.

5.2.4 Default parameters

Model name	Parameters
Multinomial Naive Bayes	scikit learn's default
K-Neighbours	K = # languages
Decision Trees	max_depth = 100
Random Forest	max_depth = 100, n_estimators = max_features = # languages
Ada-Boost	scikit learn's default
Extra Trees	max_depth = 100, n_estimators = max_features = # languages

Table 45: Default parameters

5.2.5 Observations

As it was noted in the "Analysis of Results" in both experiments, the classifier that was the most performant was ExtraTrees with a big number of estimators. However, it uses way more computational resources than this same classifier with fewer estimators and slightly more features. To be concrete, around 2 orders of magnitude less CPU Wall time while the accuracy difference is less than 3%.

As it can be seen in the results, using the multi-character token approach yields around 7% more accuracy in our tests. This can be explained because longer tokens make easier to discern which programming language the source code is written in.

We decided to layout the experiments with "default" parameters first, that is, parameters that were chosen arbitrarily so we could measure the performance of the classifiers that had multiple tunable parameters in a more easy way, as well as having some baseline with non-optimal parameters.

Some interesting albeit pretty obvious result of this approach to source code identification can be observed in the confusion matrices. Failures in classification usually happen in programming languages with a high degree of similarity. For example, in the multi-character token experiment with default parameters, we can observe that C++ is often mistaken with C. JavaScript is confused with Scheme pretty often as well with PHP. The reserved words in C++ and C are knowingly similar, but we can find similarities in programming language just by looking at where our classifier missed its prediction. That could be used for automatic grouping of languages with similar characteristics.

5.2.6 Performance

In order to measure the performance of the `ExtraTreesClassifier` classifier, we have passed a thousand Python source code files from Rosetta Code. We want to measure what is the initial start-up latency caused by the model and interpreter load, as well as some statistics for the classification of these files.

- **Start up time (latency)**: For the start up time of the Python interpreter and library loading, we have executed the classifier 10 times and computer the time difference between the execution of the command in the terminal and the moment in when everything was already loaded:

Startup latency of the classifier	
Min	0.15s
Max	0.19s
Avg	0.16s

Table 46: Startup latency of the classifier

  The results here were really consistent, with most of the data points being equal to the minimum. It is important to take into account that the Python startup time baseline, an empty text file, it is around 0.03s on this machine.

- **Throughput**: We wanted to experiment with the number of classifications we could do per unit of time. We loaded all the libraries once and performed the test in a loop, so the total load time should be negligible as shown above. In the following table, we show some basic statistics on the code classification.

Time spent per file	
Min	0.14s
Max	0.58s
Average	0.14s
Stdev	0.01s

Table 47: Time spent per file

  Dividing the amount of experiments, 1000, by the sum of times of each experiment, 142.61, yields a throughput of 7.01 file classifications per second. This number is pretty low, as we would like to have a higher vertical scalability.

  After some basic profiling, we realised we were loading the model once per iteration, which is not necessary and it is a waste of resources. After optimising the code a bit, we managed to get the sum of the runtime of all the experiments down to 106.6s, which would be 9.38 classified files/second.

5.2.7 Model weight

In this section we show the weights of the generated models for the most accurate experiment. These are relevant as they are going to be loaded in memory and they have to reach the machines in which the classification is going to be performed, hence, the lower the file size, the better.

Classifier	File size
AdaBoostClassifier.model	52K
DecisionTreeClassifier.model	411K
ExtraTreesClassifier.model	17M
KNeighborsClassifier.model	22M
MultinomialNB.model	63K
RandomForestClassifier.model	8.1M
SGDClassifier.model	3.0K

Table 48: Trained models weight for the most accurate experiment of RandomForestClassifier

5.2.8 Future directions in performance

As described in the previous section, we managed to classify around 10 files per second with our current classifier. For environments which require a higher throughput, this could be scaled horizontally, that is, by using more machines to process files in parallel. We would like to improve the performance of a single node. This would potentially decrease the cost of running this classifier, as less machines will be needed, and the operational costs will go down as well, as there will be less hardware to take care of. This kind of scaling is called vertical scaling because it usually happens in a single node.

We didn't fully investigate how to improve the performance of our classifier, as it was a bit outside of the scope of the project. That being said, we find it is a really important part of any production system, so we had a look at possible ways to improve the efficiency of the classifier.

One of the ideas was to compile our model to a language closer to the metal, like C. This would be highly beneficial because it would allow us to continue to use Python for prototyping, with all the advantages it carries, but have lower startup time and a potentially higher throughput.

The other added benefit of compiling the model to a language like C or C++ is that most languages offer the possibility to create native extensions in which the models could be reused among languages. One library we found that compiles the scikit model to C or Java code is is [36].

5.3 Another approach: Syntax checkers

The approach described here is simple, effective, and inefficient: Many language's compilers/interpreters have an option that just checks the syntax of the program without executing the code, so why not running all of them over some piece of code and just return the language whose binary said that the syntax was correct? Unfortunately, it has some important disadvantages.

Some of the problems that this approach has are:

- Not every interpreter/compiler has the built-in option to only check the syntax
- Running all those interpreters for each file doesn't scale well, as we have to run *n* syntax checkers. Some of them take a bit to start up, and the CPU consumption would be pretty high.
- This approach wouldn't work with code snippets. E.g: Some Java method which is not inside of a class.
- There exists the possibility of more than one syntax checker validating the syntax. Sometimes it's just a matter of configuration, like a C++ compiler with C compatibility enabled.
- Executing those processes implies running something like Unix's fork + exec or a similar call like the more efficient process_spawn. Those system calls have a pretty big overhead.

An approach that could be more efficient and that we think it could be interesting to explore in future work would be doing a "union" of grammars, in which there's at least one finishing state for each language. That would be more efficient, but extremely more difficult to maintain, as we would have to have up to date grammars for all the languages we would like to test, develop, etc. Moreover, we think this would be possible but we are not sure if this is the case.

Note: Having those syntax checkers report the syntax is correct is not guarantee that the program would be able to run. Not the biggest deal, as with the machine learning approach we cannot even verify if the syntax is correct and we don't care about this, but in some languages, especially interpreted ones, some source code whose syntax is valid might not run. For example, lexical scope: accessing a variable that hasn't been defined before it's not going to surface with this approach.

6 Conclusion

In this project we have built a programming language identifier using machine learning techniques. We have evaluated the training time and accuracy of different machine learning models and tuned their parameters.

All the code was developed in Python, which allowed us to iterate rapidly and leverage the wide variety of high-quality scientific libraries. We were already familiar with the language, which was highly beneficial to get up to speed quickly.

This work required knowledge of basic programming, the Python programming language, machine learning, and software engineering practices, among others.

We managed to obtain an accuracy slightly higher than 80% in one of the best classifiers trained for 14 different languages, while keeping the training time reasonably low, which we consider this a decently successful result.

Some of the future work directions that we find worth exploring are: using more compiler approaches for this problem, such as the aforementioned union of grammars, as well as trying other machine learning approaches like deep learning algorithms. Another idea we had that we couldn't implement because of time constraints would be to apply a model like word2vec [33] to source code.

7 Annex

7.1 User manual

This section details how to run several basic tasks with the developed code, such as installing its dependencies, training the models, and classifying a file. In this project, this section is also the programmer's manual, as the programmer and the user are the same person.

7.1.1 Installing Homebrew[28]

```
$ /usr/bin/ruby -e \$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

7.1.2 Installing Python using Homebrew

```
$ brew install python3
```

7.1.3 Installing the dependencies of the project

Install the Python dependencies for the project.

```
$ script/bootstrap
```

Figure 15: Installing the dependencies

7.1.4 Model training

Train the models with the configuration specified in `config.py`:

```
$ python3 train.py
```

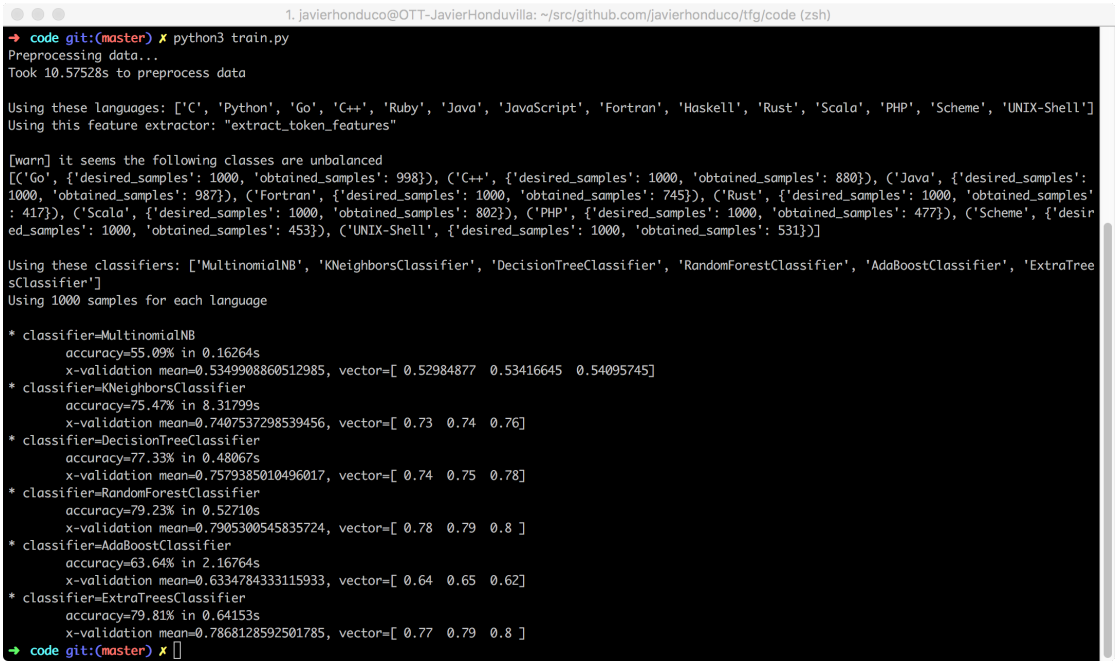

Figure 16: Training the models

7.1.5 Classifying a file from this same repo

Use the default trained model to classify some provided source code file.

```
$ python3 classify.py <file>
```

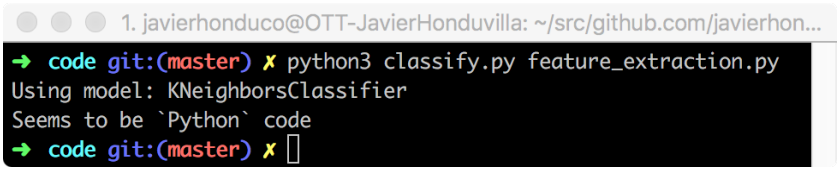

Figure 17: Successfully classifying a Python file

7.2 One character tokens

; ({ [: . | $ & < > # ! ? @

7.3 Multi character tokens

'' * , - -- -< -<< -> :: ; <- = => > ? @ BEGIN END False None Self True [[[[|, |] \]]] _ __FILE__ __LINE__ __halt_compiler ` abstract access alias alignof and array as assert assign auto backspace become begin bkpt block boolean box break byte call callable case catch chan char class clone close common cond cons-stream const continue crate data debugger declare def default default-object? defer define define-integrable define-macro define-structure define-syntax defined? del delay delete deriving deriving instance die dimension do done double echo elif else elseif elsif empty end enddeclare endfile endfor endforeach endif endswitch endwhile ensure entry enum equivalence esac eval except exit extends extern external fallthrough false family fi final finally float fluid-let fn for forall foreach foreign format from func function global go goto hiding if impl implements implicit import in in-package include include_once infix infixl infixr inquire instance instanceof insteadof int interface intrinsic is isset lambda let let* let-syntax letrec list local-declare long loop macro make-environment map match mdo mod module move mut named-lambda namespace native new newtype next nil nonlocal not null of offsetof open or override package parameter pass pause print printf println priv private proc program protected pub public pure qualified quasiquote quote raise range read rec redo ref register require require_once rescue retry return rewind rewrite save scode-quote select self sequence set! short signed sizeof static stop strictfp struct subroutine super switch synchronized the-environment then this throw throws time to trait transient true try type typedef typeof unassigned? undef union unless unsafe unset unsigned unsized until use using-syntax val var virtual void volatile when where while with write xor yield { | }

7.4 MIT License template

Copyright {YEAR} {COPYRIGHT HOLDER}

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Bibliography and References

[1] *A Short History of Git*. URL: https://git-scm.com/book/en/v2/Getting-Started-A-Short-History-of-Git.

[2] *A Whole New Code Search*. URL: https://github.com/blog/1381-a-whole-new-code-search.

[3] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers: principles, techniques, and tools*. Vol. 2. Addison-wesley Reading, 2007.

[4] *All the open source code in GitHub now shared within BigQuery: Analyze all the code!* URL: https://medium.com/google-cloud/github-on-bigquery-analyze-all-the-code-b3576fd2b150.

[5] *Amount of profanity in git commit messages per programming language*. URL: http://andrewvos.com/2011/02/21/amount-of-profanity-in-git-commit-messages-per-programming-language.

[6] *Apache Lucene - Apache Lucene Core*. URL: https://lucene.apache.org/core/.

[7] Dimitar Asenov, Otmar Hilliges, and Peter Müller. "The Effect of Richer Visualizations on Code Comprehension". In: *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. CHI '16. San Jose, California, USA: ACM, 2016, pp. 5040–5045. ISBN: 978-1-4503-3362-7. DOI: 10.1145/2858036.2858372. URL: http://doi.acm.org/10.1145/2858036.2858372.

[8] *BitKeeper*. URL: http://www.bitkeeper.org/.

[9] *cPython's source code*. URL: https://github.com/python/cpython/blob/master/Lib/re.py#L334.

[10] *cPython's tokenizer*. URL: https://github.com/python/cpython/blob/master/Parser/tokenizer.c.

[11] Juriaan Kennedy van Dam. "Identifying Source Code Programming Languages through Natural Language Processing". Master's thesis. Amsterdam, The Netherlands: Universiteit van Amsterdam, Jan. 2016.

[12] Amit Deshpande and Dirk Riehle. "The Total Growth of Open Source". In: *Open Source Development, Communities and Quality: IFIP 20th World Computer Congress, Working Group 2.3 on Open Source Software, September 7-10, 2008, Milano, Italy*. Ed. by Barbara Russo et al. Boston, MA: Springer US, 2008, pp. 197–209. ISBN: 978-0-387-09684-1. DOI: 10.1007/978-0-387-09684-1_16. URL: https://doi.org/10.1007/978-0-387-09684-1_16.

[13] Amit Deshpande and Dirk Riehle. "The Total Growth of Open Source". In: *Open Source Development, Communities and Quality: IFIP 20th World Computer Congress, Working Group 2.3 on Open Source Software, September 7-10, 2008, Milano, Italy*. Ed. by Barbara Russo et al. Boston, MA: Springer US, 2008, pp. 197–209. ISBN: 978-0-387-09684-1. DOI: 10.1007/978-0-387-09684-1_16. URL: https://doi.org/10.1007/978-0-387-09684-1_16.

[14] *Farewell to Google Code*. URL: https://opensource.googleblog.com/2015/03/farewell-to-google-code.html.

[15] *FB MySQL*. URL: https://www.mysql.com/customers/view/?id=757.

[16] *Fine Free File Command*. URL: http://www.darwinsys.com/file/.

[17] *Git*. URL: https://git-scm.com/.

[18] *GitHub Archive*. URL: https://www.githubarchive.org/.

[19] *GitHub. Build software better, together*. URL: https://github.com/.

[20] *GitHub file view*. URL: https://github.com/github/scientist/blob/master/lib/scientist/observation.rb.

[21] *GitHub org view*. URL: https://github.com/github.

[22] *GitHub search drilldown*. URL: https://github.com/github/linguist/search?utf8=%5C%E2%5C%9C%5C%93&q=%5C%22module%5C%22&type=.

[23] *github/linguist: Language Savant. If your repository's language is being reported incorrectly, send us a pull request!* URL: https://github.com/github/linguist/.

[24] *Gitlab CE stats*. URL: https://gitlab.com/gitlab-org/gitlab-ce/graphs/master/charts.

[25] *GNU's Bison*. URL: https://www.gnu.org/software/bison/.

[26] *Google MySQL*. URL: https://www.mysql.com/customers/view/?id=555.

[27] Daniel Graziotin et al. "On the Unhappiness of Software Developers". In: *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*. EASE'17. Karlskrona, Sweden: ACM, 2017, pp. 324–333. ISBN: 978-1-4503-4804-1. DOI: 10.1145/3084226.3084242. URL: http://doi.acm.org/10.1145/3084226.3084242.

[28] *Homebrew — The missing package manager for macOS*. URL: https://brew.sh/.

[29] J. D. Hunter. "Matplotlib: A 2D graphics environment". In: *Computing In Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55.

[30] *iTerm2 - macOS Terminal Replacement*. URL: https://www.iterm2.com/.

[31] Jyotiska Nath Khasnabish et al. "Detecting Programming Language from Source Code Using Bayesian Learning Techniques". In: *Machine Learning and Data Mining in Pattern Recognition: 10th International Conference, MLDM 2014, St. Petersburg, Russia, July 21-24, 2014. Proceedings*. Ed. by Petra Perner. Cham: Springer International Publishing, 2014, pp. 513–522. ISBN: 978-3-319-08979-9. DOI: 10.1007/978-3-319-08979-9_39. URL: https://doi.org/10.1007/978-3-319-08979-9_39.

[32] *Languages supported by Linguist*. URL: https://github.com/github/linguist/blob/c8171322f502f1044b700925b0104b41517ebcc7/lib/linguist/languages.yml.

[33] Tomas Mikolov et al. "Efficient estimation of word representations in vector space". In: *arXiv preprint arXiv:1301.3781* (2013).

[34] *MIME types - HTTP — MDN*. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_types.

[35] *monotone*. URL: https://www.monotone.ca/.

[36] Darius Morawiec. "sklearn-porter". Transpile trained scikit-learn estimators to C, Java, JavaScript and others. URL: https://github.com/nok/sklearn-porter.

[37] Lorraine Morgan and Patrick Finnegan. "Benefits and Drawbacks of Open Source Software: An Exploratory Study of Secondary Software Firms". In: *Open Source Development, Adoption and Innovation: IFIP Working Group 2.13 on Open Source Software, June 11–14, 2007, Limerick, Ireland*. Ed. by Joseph Feller et al. Boston, MA: Springer US, 2007, pp. 307–312. ISBN: 978-0-387-72486-7. DOI: 10.1007/978-0-387-72486-7_33. URL: https://doi.org/10.1007/978-0-387-72486-7_33.

[38] *Open Source Search Analytics · Elasticsearch*. URL: https://www.elastic.co/.

[39] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[40] *Python AST Visualizer*. URL: https://vpyast.appspot.com/.

[41] *Ragel State Machine Compiler*. URL: http://www.colm.net/open-source/ragel/.

[42] *RE2 is a fast, safe, thread-friendly alternative to backtracking regular expression engines like those used in PCRE, Perl, and Python. It is a C++ library*. URL: https://github.com/google/re2.

[43] *Regular Expression Matching with a Trigram Index or How Google Code Search Worked*. URL: https://swtch.com/~rsc/regexp/regexp4.html.

[44] *Rosetta Code*. URL: http://rosettacode.org/.

[45] *Ruby on Rails — A web-application framework that includes everything needed to create database-backed web applications according to the Model-View-Controller (MVC) pattern*. URL: http://rubyonrails.org/.

[46] *Ruby Programming Language*. URL: https://www.ruby-lang.org/en/.

[47] *SourceForge - Download, Develop and Publish Free Open Source Software*. URL: https://sourceforge.net/.

[48] *The leading product for integrated software development - GitLab — GitLab*. URL: https://gitlab.com/.

[49] *The Lex And Yacc Page*. URL: http://www.colm.net/open-source/ragel/.

[50] Secil Ugurel, Robert Krovetz, and C. Lee Giles. "What's the Code?: Automatic Classification of Source Code Archives". In: *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '02. Edmonton, Alberta, Canada: ACM, 2002, pp. 632–638. ISBN: 1-58113-567-X. DOI: 10.1145/775047.775141. URL: http://doi.acm.org/10.1145/775047.775141.

[51] *vim*. URL: http://www.vim.org/.

[52] S. van der Walt, S. C. Colbert, and G. Varoquaux. "The NumPy Array: A Structure for Efficient Numerical Computation". In: *Computing in Science Engineering* 13.2 (Mar. 2011), pp. 22–30. ISSN: 1521-9615. DOI: 10.1109/MCSE.2011.37.

[53] *Welcome to Python.org*. URL: https://www.python.org/.

8 Glossary

- **AST**: An Abstract Syntax Tree is a data structure that is used to represent the structure of a piece of source code. It is emitted in the parsing stage.
- **CPU Time**: What was the total amount of time between the start and the end of an event aggregating all threads that were doing work for this task.
- **Confusion matrix**: Matrix that is often used in machine learning to identify the accuracy of some classifier. In both axes, we can find all the classes that we have trained for. In each cell, we have the number of predictions that we had for class c_j that the classifier got for class c_i. When $i = j$, that means the prediction was correct. Otherwise, it means the prediction was wrong.
- **Cross validation**: Technique in statistics that consists on splitting the training set in two sets. One of them is used for training the model, and the other one to test it. The advantage of this approach is that training bias is decreased and the assessment of the model's performance is more accurate.
- **ElasticSearch**[38]: an open source search indexing engine that builds atop of the Apache Lucene[6] project.
- **Free Software**: Software that guarantees user freedom.
- **Git**: a popular Source Control Management system that is used by many companies and individuals and that many Git hosting companies leverage.
- **GitHub**[19]: Git repository hosting solution.
- **GitLab**[48]: Git repository hosting solution which is open source.
- **Horizontal Scaling**: Scaling some software to run distributed in several nodes, so the load is spread out.
- **Lexer**: Component of a parser which receives the raw input text and split it into tokens.
- **Library**: Set of reusable source code files.
- **Linguist**: Ruby library created by GitHub to detect the programming language in which a given file is written. It is used in production by them and by other companies, such as GitLab.
- **MIME type**: Multipurpose Internet Mail Extensions is a standardised way of indicating the format and structure of a document [34].
- **Open Source Software**: Software for which source code it is available for inspection.

- **Parser**: Component of a parser which receives tokens and generates some Abstract Syntax Tree or does some other processing with it, like syntax-driven translation.
- **Profiling**: Set of techniques that enable programmers to see the impact of some fragment of code on some of the computer's resources, like CPU or memory
- **Python**[53]: is an interpreted object-oriented programming language
- **Repo**: Short version of Repository.
- **Repository**: Set of files stored under a SCM.
- **Rosetta Code Project**[44]: Project that has the mission of writing examples of most popular algorithms in every programming language.
- **Ruby**[46]: Interpreted, object-oriented scripting programming language that was created in Japan in 1995 and made popular by the Ruby on Rails web framework [45].
- **Scikit-learn**[39]: Machine learning library for Python.
- **Shebang**: in UNIX operating systems, the `#!` at the beginning of a script that refers to the binary with which the script iself can be executed.
- **Source Control Management (SCM)**: Piece of software that enables the creation of self-contained repositories of code or any other files that enable history tracking, authoring of changes, etc. Most of them also enable smoother collaboration work-flows with other people.
- **Token**: List of characters which is atomic and hence cannot be broken down into smaller chunks. These are generally the output of a Lexer.
- **Vertical Scaling**: Scaling some software to run more efficiently on a single node.
- **Wall Time (of a process)**: Time passed since the beginning of one event, such as an experiment, until the end, no matter how many OS threads were involved in producing the result.
- **file(1)**[16]: UNIX utility that given a file it detects which kind of file it looks like it may be.
- **pip**: a popular Python's package manager used to install dependencies.